Pythran: Crossing the Python Frontier

Serge Guelton
Institut Mines-Télécom
Bretagne

Editors:
Konrad Hinsen,
konrad.hinsen@cnrs.fr;
Matthew Turk,
matthewturk@gmail.com

Use of the Python language in scientific computing has always been characterized by the coexistence of interpreted Python code and compiled native code, written in languages like C or Fortran. To relieve scientists from the burden of using lower-level languages, tools such as Cython and Numba were developed to ease the transition from interpreted to native code in the course of code optimization. In this column, I take a fresh look at this problem and introduce Pythran, a new optimization tool designed with the goal of efficiently handling unmodified Python code.

The native/interpreted frontier problem is at the core of Python's design. Every time I look at it from a performance point of view, this problem arises.

Python was initially designed as a glue language, bridging the world of low-level, efficient but difficult-to-use libraries, and easy-to-learn, scriptable languages. This implies a conversion between native data and generic Python objects, the "PyObject."

As a reference, the following naïve implementation of the Rosenbrock function (a function commonly used to benchmark optimization functions[1]) creates $\mathcal{O}(n)$ intermediate objects, wrapping a native value. This is terribly inefficient.

```
def rosen_explicit_loop(x):
  s = 0.
  n = s.shape
  for i in range(0, n - 1):
    s += 100. * x[i + 1] - x[i] ** 2.) ** 2. + (1 - x[i]) ** 2
  return s
```

As time passed, Python was adopted by the scientific community. But in this world of number crunching, data conversion is a loss of time, so the NumPy array[2] was designed to avoid exactly that issue. A NumPy array provides a wrapper around plain flat memory. It can be converted between native and interpreted code at almost no cost, in $\mathcal{O}(1)$. Thanks to this abstraction, data can freely cross the Python frontier.

Although NumPy arrays are a great improvement, they still suffer from the Python frontier, as any function call made from the interpreted world, even if executed in the native world, needs to provide a result in the interpreted one. Consequently, every complex expression is split in small efficient computations, with data moving back and forth between the Python and native world.

Copublished by the IEEE CS and the AIP
1521-9615/18/$33 ©2018 IEEE

For instance, the following code snippet creates seven temporary arrays. This is much better than the previous example but still less efficient than a native loop.

```
import numpy as np
def rosen_numpy(x):
  return np.sum(100.0*(x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)
```

Higher order functions, that is, functions that take another function as an argument, suffer from the frontier problem even when compiled to native code, because their arguments can be interpreted functions, which results in extra cross-frontier conversions. This simple 1D filter, whose implementation runs native code, still performs $\mathcal{O}(n\times m)$ conversion.

```
from scipy.ndimage import generic_filter1d

def filter(iline, oline):
  oline[:] = iline[:-4] + 2 * iline[2:-2] + 3 * iline[4:]

generic_filter1d(data, filter, 5)
```

(C)Python, the historically first (and still most widely used) implementation of Python written in C, proposes a simple way to solve all these issues: write a native Python module that does the job in C and use the Python C API to expose the result of the computation. If you are skilled enough, this can lead to very efficient code, where you can unleash vectorization, parallelization, and all the optimization techniques applicable at that level. But you lose in productivity.

TRADING READABILITY FOR PERFORMANCE

Based on the fact that scientific users are more interested in the Python syntax than in the dynamic behavior when writing scientific kernels, solutions such as Cython[3] and Numba[4] emerged. Even if the implementations radically differ (the former is an ahead-of-time compiler, the latter is a just-in-time compiler), they share a common approach: translating Python code into native code using a compiler and type information gathered from the declaration or the runtime environment in order to remove part of the dynamic behavior of Python.

This proved to be a valid approach, as the user base quickly grew. These two tools are now important elements of the Python scientific ecosystem. Yet, there is a price to pay, as illustrated by this piece of Cython code:

```
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def rosen_cython(double[:] x):
  cdef int n = x.shape[0]
  cdef int i
  cdef double s = 0.

  for i in range(0,n-1):
    s += 100. * (x[i + 1] - x[i] ** 2.) ** 2. + (1 - x[i]) ** 2
  return s
```

Several elements are disturbing with this code:

- The user needs to learn an extra language. Not knowing about the “cython” decorator results in less efficient code.
- The code is much less abstract and concise than the original Numpy version.
- The code is no longer polymorphic: to produce a single precision floating point version, it needs to be duplicated with all references to “double” replaced by “float.”

Compared to the performance boost, the price to pay is not crippling, but can we improve it?

INTRODUCING PYTHRAN

Pythran[5] is an ahead-of-time compiler for Python and is built around four core principles:

1. *Backward compatibility.* Any valid input for Pythran is also a valid input for Python.
2. *Type agnosticism.* Pythran does not use extra type information during its compilation process. Analyses and optimizations, as well as C++ code generation, are independent from the actual function parameter type
3. *High level.* Scientists tend to think in terms of high-level transformation on data arrays. Supporting this programing style in addition to explicit loops is mandatory.
4. *Pure native.* Once the conversion step is done, all computations are done in native code, without any reference to the Python C API.

As a matter of comparison, Cython does not support principles 1, 2, or 3 and has optional support for 4. Numba supports 1 and optionally 4.

These principles are illustrated by the Pythran version of the "rosen" function:

```
#pythran export rosen_pythran(float32[])
#pythran export rosen_pythran(float64[])
import numpy as np
def rosen_pythran(x):
  return np.sum(100.0*(x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)
```

As a consequence of 1, the Python code is the same as the Numpy version. As a consequence of 2, the compiler first translates this function into a generic C++ meta-function without using the type annotations. As a consequence of 3, there is no need for the user to reimplement the "numpy.sum" built-in or the various Numpy "ufunc" involved in the computation through an explicit loop. Finally, as a consequence of 4, the compiler *instantiates* this meta-function for the two given native types, then creates a small wrapper to make it callable from Python.

Let's compare the number of source lines of code for the three approaches:

- NumPy: 3
- Cython: 10
- Pythran: 5

Nothing surprising here: writing extra annotations and explicit loops make Cython more verbose.

Now let's look on the performance side. The Cython and Pythran version are compiled with GCC version 6.3, without activating autoparallelization or autovectorization, just the default flags (both fall back to "-O2" with a few extra flags, different for each compiler).

```
>>> import numpy as np
>>> data = np.random.random(100000)

>>> %timeit rosen_numpy(data)
359 µs ± 25.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %timeit rosen_cython(data)
181 µs ± 5.74 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

>>> %timeit rosen_pythran(data)
143 µs ± 4.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

This is a very simple micro-benchmark. However, there is still a factor of two between the execution time of the NumPy and native versions. This difference is explained by the boundary problem: intermediate arrays are created by NumPy each time an operation is evaluated, whereas those copies are avoided in the case of Cython and Pythran.

Pythran is slightly faster than Cython, but they are within the same magnitude order, and they do not use the same back-end language nor the exact same compilation flags.

A More Complex Example

The following Python code uses NumPy to perform a simple laplacian convolution. It takes advantage of polymorphism to handle both grayscale images (2D arrays of floating point numbers) and color images (3D arrays of floating point images).

```
  def convolve_laplacian(image):
    out_image = np.abs(4*image[1:-1,1:-1] - image[0:-2,1:-1] - image[2:,1:-1] -
image[1:-1,0:-2] - image[1:-1,2:])
    # normalize
    valmax = np.max(out_image.flat)
    valmax = max(1.,valmax)+1.E-9
    out_image *= 1./valmax
    return out_image
```

On one hand, the code accepts both 2D and 3D images, therefore, the Cython or Numba user needs to write the function body twice. On the other hand, the Pythran user only needs to write the following comments and to compile the code to get a maintainable, polymorphic function.

```
  #pythran export convolve_laplacian(float64[][])
  #pythran export convolve_laplacian(float64[][][])
```

That's a direct benefit of principle 2!

About Capsules

As a workaround for higher-level functions written in native code, like "scipy.ndimage.generic_filter1d," which keep on calling interpreted code, the concept of "PyCapsule" has been introduced in the Python C API. It is a very simple "PyObject" that wraps a raw C pointer, an extra text field and some optional user data. The goal of this capsule is to be able to cross the boundary within the capsule at no cost. Just like an "ndarray"! However, it has no specific API, apart from retrieving the raw pointer and the associated string.

This capsule concept is still a big step forward, as it allows one to use C function pointers generated in some way as a parameter of a function like "scipy.ndimage.generic_filter1d." The latter then accesses the raw pointer and calls it, without extra boundary crossing. It is a way for different native code to communicate through the interpreted layer at no cost.

Thanks to principle 4, it's easy for Pythran to generate a 100 percent native function from Python code and put it into a capsule instead of adding the conversion layer.

Translating from interpreted, dynamic code to statically compiled code indeed brings a performance improvement, but it is not enough. Advanced compiler techniques can provide much more than this. Following principle 1, it first takes the original source code, adds an annotation plus an extra conversion step to be able to use Numpy array notations:

```
  #pythran export capsule cfilter(float64*, int64, float64*, int64, int*)
  #pythran export filter(float64[], float64[])
  from numpy.ctypeslib import as_array
  def filter(iline, oline):
    oline[:] = iline[:-4] + 2 * iline[2:-2] + 3 * iline[4:]

  def cfilter(iline_data, iline_size, oline_data, oline_size, user_data):
    iline = as_array(iline_data, iline_size)
    oline = as_array(oline_data, oline_size)
    filter(iline, oline)
    return 0
```

Using this approach, it is possible to call the function from the capsule, leading to native higher order function calling native code, without any Python conversion, which leads to better performance as showcased by the code snippet below.

```
>>> from scipy import LowLevelCallable
>>> import numpy as np
>>> data = np.random.random((1000,1000))
>>> %timeit generic_filter1d(data, filter, 5)
4.53 ms ± 84.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> cfilter = LowLevelCallable(cfilter, signature='int (double *, intptr_t, dou-
ble *, intptr_t, void *)')
>>> %timeit generic_filter1d(data, cfilter, 5)
3.27 ms ± 89.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

ABOUT THE GIL

One of the well known limitations of (C)Python is the lack of concurrency support. It does support multithreading, for example, through the "Threading" module, but many interpreted operations involve taking and releasing the *Global Interpreter Lock*, a global mutex used mostly (but not solely) for thread-safe memory management.

When code that does not reference any Python object is used, the GIL can be released. Thanks to property 4, this is *always* the case in Pythran: once Python objects are converted into Pythran ones, there is no further interaction with the Python C API, so the GIL is released. As a consequence, a code like the following runs faster as cores get added, provided "pythranized" is a function compiled with Pythran: once each thread is in the native section, no lock is involved.

```
threads = [Thread(target=pythranized, args=[i]) for i in range(1, core_count)]
for thread in threads:
  thread.start()
for thread in threads:
  thread.join()
```

MORE ABOUT CONCURRENCY

OpenMP[6] is a common way to exploit parallelism in statically compiled languages. In C, OpenMP is used through compiler directives, "#pragma," and in Fortran, it is used through code comments. The nice property of this approach is backward-compatibility: a compiler can ignore them and still compile a valid sequential program. Annotations turn a sequential program into a parallel one. Note that in some cases, the semantics differ when the OpenMP directives are ignored; just think about parallel reduction that assumes floating point arithmetic is associative.

Pythran follows the same philosophy as a consequence of principle 1. Because it uses C++ as an intermediate back end, it is possible to integrate OpenMP directives in the Pythran front end and keep them until C++ code generation, while making all Pythran optimizations aware of them.

That's actually how it is implemented in Pythran, making the following code a completely valid Python program, a valid Pythran sequential program, *and* a valid Pythran parallel program that checks the longest Collatz sequence a given bound:[7]

```
def collatz(n):
  if n == 1:
    return 0
  if n & 1:
    return 1 + collatz(3 * n + 1)
  else:
    return 1 + collatz(n / 2)

def euler14(n):
  m = 0
```

```
#omp parallel for reduction (max:m)
for i in range(1, n):
  m = max(collatz(i), m)
return m
```

LIMITATIONS

The properties that guide Pythran development also have a few negative impacts:

- Only a Python subset is supported. Some Python concepts are difficult to represent in a polymorphic way. User classes (difficulty to type implicit state) and mutable global variables (their type depend on usage of polymorphic functions) belong to that category.
- Only a limited subset of Python modules is supported. The lack of a mixed mode where the Python C API is called as a fallback implies that every imported module should be rewritten in C++. This is not realistic, so we focus on core scientific modules like "math," "random," or "numpy." The list of supported modules and functions is available in the online Pythran documentation (pythran.readthedocs.io).
- Backward compatibility prevents the explicit usage of C or C++ library from Pythran program, whereas the internal machinery would make it a feasible task.
- As Pythran relies on a third party C++ compiler for its back end, performance also depends on the capacity of this compiler to exploit the code generated by Pythran.

CONCLUSION

This article highlights the impact of Python's frontier problem on scientific computing, and introduces the core principles of Pythran, an ahead-of-time compiler for scientific Python. It showcases how Pythran can help pass the native/interpreted boundary, providing significant speedups over high-level Numpy implementations without changing the original implementation.

ACKNOWLEDGMENTS

As a final note, the author would like to thank William Gouzier for his careful review, and the OpenDreamKit project for its funding of Pythran development.

REFERENCES

1. "Rosenbrock Function," Wikipedia, 10 Dec. 2017; en.wikipedia.org/wiki/Rosenbrock_function.
2. S. van der Walt, S.C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, 2011, pp. 22–30.
3. S. Behnel et al., "Cython: The Best of Both Worlds," *Computing in Science & Engineering*, vol. 13, no. 2, 2011, pp. 31–39.
4. "Numba," 2017; numba.pydata.org.
5. S. Guelton et al., "Pythran: Enabling Static Optimization of Scientific Python Programs," *Computational Science & Discovery*, vol. 8, no. 1, 2015; iopscience.iop.org/article/10.1088/1749-4680/8/1/014001/meta.
6. "OpenMP," 2012; openmp.org.
7. "Longest Collatz Sequence," Project Euler; projecteuler.net/problem=14.

ABOUT THE AUTHOR

Serge Guelton is an associate researcher at Institut Mines-Télécom Bretagne. His research interests include compilers, vectorization, and HPC. Guelton received a PhD in compilation from Télécom Bretagne. Contact him at serge.guelton@telecom-bretagne.eu.